This is the final peer-reviewed accepted manuscript of:

G. Tagliavini, G. Haugou, A. Marongiu and L. Benini, "ADRENALINE: An OpenVX Environment to Optimize Embedded Vision Applications on Many-core Accelerators," 2015 IEEE 9th International Symposium on Embedded Multicore/Many-core Systems-on-Chip, Turin, 2015, pp. 289-296

The final published version is available online at:

http://dx.doi.org/10.1109/MCSoC.2015.45

Rights / License:

The terms and conditions for the reuse of this version of the manuscript are specified in the publishing policy. For all terms of use and more information see the publisher's website.

This item was downloaded from IRIS Università di Bologna (https://cris.unibo.it/)

When citing, please refer to the published version.

ADRENALINE: an OpenVX environment to optimize embedded vision applications on many-core accelerators

Giuseppe Tagliavini
University of Bologna, Italy
Email: giuseppe.tagliavini@unibo.it

Germain Haugou
ETH Zurich, Switzerland
Email: haugoug@iis.ee.ethz.ch

Andrea Marongiu and Luca Benini
University of Bologna, Italy
ETH Zurich, Switzerland
Email: {a.marongiu, luca.benini}@iis.ee.ethz.ch

***Abstract*—The acceleration of Computer Vision algorithms is an important enabler to support the more and more pervasive applications of the embedded vision domain. Heterogeneous systems featuring a clustered many-core accelerator are a very promising target for embedded vision workloads, but the code optimization for these platforms is a challenging task. In this work we introduce ADRENALINE[1], a novel framework for fast prototyping and optimization of OpenVX applications for heterogeneous SoCs with many-core accelerators. ADRENALINE consists of an optimized OpenVX run-time system and a virtual platform, and it is intended to provide support to a wide range of end users. We highlight the benefits of this approach in different optimization contexts.**

Keywords*—*embedded vision, OpenVX, virtual platform, accelerator

I. Introduction

In recent years the industrial community has strongly focused on embedded vision technologies, which aim at incorporating Computer Vision (CV) capabilities into a wide range of embedded systems. Typical examples of these applications are gesture tracking, smart video surveillance, advanced driver assistance systems (ADAS), augmented reality (AR), and many others. In this context the acceleration of CV functions is increasingly considered a must, since many of these applications are computationally intensive.

The increasing adoption of vision capabilities within embedded systems requires both high performance and energy efficiency. Towards these goals, architectural heterogeneity is being more and more adopted to design embedded vision systems, where a multi-core host processor is coupled to programmable many-core accelerators [5] [22] [16] [4] [10]. These accelerators provide tens to hundreds of small processing units, connected to a shared on-chip memory via a low-latency, high-throughput interconnection. Heterogeneous systems based on many-core programmable accelerators have the potential to dramatically increase the peak performance/Watt, but they come at the price of increased programming complexity. The role of programming models aimed at simplifying this task becomes thus paramount. In the last years the Khronos consortium [2] has been very active in proposing new programming standards for heterogeneous platforms. For many years OpenCL [17] has been the best representative of this category, and nowadays it is widely supported by major SoC vendors. OpenCL aims at defining a standard interface for cross-platform heterogeneity exploitation, but it exposes a very low-level API which requires both a significant effort by the programmers and an intimate knowledge of the target hardware. To increase the productivity of application designers, it is important that the programming model exposes high-level constructs for the exploitation of parallel and heterogeneous resources. In this way, experts of the application can focus on its partitioning and deployment, without the need for expertise on the hardware details. This is particularly true in the CV domain, where the expertise of application designers is typically on the algorithms.

OpenVX [18] has been introduced as a cross-platform standard for imaging and vision application domains, with the aim to raise significantly the level of abstraction at which CV applications should be coded. Based on a standard plain C API, it is easy to use and fully transparent to architectural details. The details of the hardware platform are hidden in the underlying run-time environment (RTE) layer. This approach enables the portability of vision applications across different heterogeneous platforms, delegating the performance tuning to hardware vendors, who provide an efficient RTE with architecture-specific optimizations.

In this work we introduce **ADRENALINE**, a novel framework for fast prototyping and optimization of OpenVX applications on heterogeneous SoCs with many-core accelerators. ADRENALINE consists of an optimized OpenVX run-time system, based on streamlined OpenCL support for a generic heterogeneous SoC template. The tool comes with a virtual platform modeling the target architecture template, which can be easily configured along several axes. The run-time system includes several optimizations for the efficient exploitation of the explicitly managed memory hierarchy adopted in the targeted SoCs, but it can be easily extended to consider other optimization opportunities. Similarly, the virtual platform can be expanded to model additional architectural blocks in a simple manner. Finally, we provide relevant use cases for our tool, showing how it can support the needs of several users.

The rest of the paper is organized as follows. Section II introduces the OpenVX programming model. In Section III we describe the architectural template we are focused on in this work. Section IV describes ADRENALINE internals in detail. Section V shows the experimental results. Section VI presents the related works. Finally, we conclude and introduce our future work in Section VII.

II. OpenVX programming model

OpenVX [18] is a cross-platform C-based Application Programming Interface (API) standard. Strongly supported by many industrial actors, OpenVX aims at enabling hardware vendors to implement and optimize low-level image processing

[1]ADRENALINE is part of the PULP project eco-system [3]. PULP is an open-source hardware platform that provides an ultra-low-power many-core accelerator template. As a tribute to the cinematographic counterpart after which the PULP was named, ADRENALINE also draws from that colourful universe of characters and situations. Have you guessed what it is yet?
Website: http://www-micrel.deis.unibo.it/adrenaline

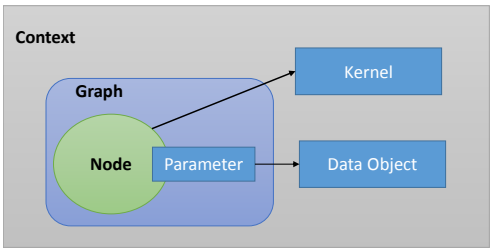

Fig. 1: OpenVX framework objects

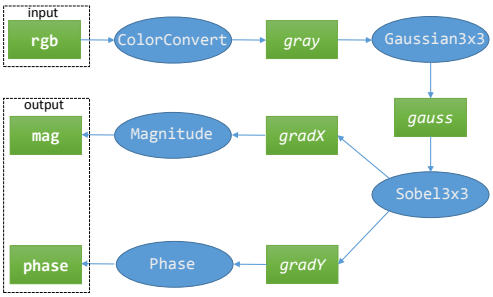

Fig. 2: Resulting graph for OpenVX sample code.

and CV primitives. Most image processing applications can be easily structured as a set of vision kernels (i.e. basic function or algorithms) that interact on the basis of input/output data dependencies. In this context, OpenVX supports a graph-oriented execution model based on a Directed Acyclic Graphs (DAGs) of kernel instances. The standard describes the software abstractions that are defined in an OpenVX execution environment, referred as *OpenVX framework*. Figure 1 depicts the framework objects and their relationship. A key point is the difference between a kernel, that is the implementation of a vision algorithm, and a node, that is an instance of a kernel inside a graph. After the creation of a framework context, an OpenVX program contains the definition of a set of *data objects*, in particular images and other helper data structures. Data objects may be declared as *virtual*. Basically, virtual data just define a dependency between adjacent kernel nodes, and are not associated with any memory area accessible by means of API functions.

OpenVX graphs are composed of one or more nodes that are added by calling node creation functions. Nodes are linked together via data dependencies, without specifying no explicit ordering. The OpenVX standard defines a library of *predefined vision kernels*, but also supports the notion of *user defined kernels* to provide new features. The standard defines 41 predefined kernels, which are fully supported in our implementation. Graphs must be *verified* before their execution, with the aim to guarantee some mandatory properties:

- Input and output requirements must be compliant to the node interface (data direction, data type, required vs optional flag).
- No cycles are allowed in the graph.
- Only a single writer node to any data object is allowed.
- Writes have higher priorities than reads.

After verification, a graph can be executed in two modes: (i) *synchronous blocking mode*, which blocks the program execution until the graph execution is completed; (ii) *asynchronous single-issue mode*, which is non blocking and enables the parallel execution of multiple graphs.

Listing 1 shows a typical example of an OpenVX program, and Figure 2 shows the corresponding DAG.

```
1  vx_context ctx = vxCreateContext();
2
3  vx_image rgb   = vxCreateImage(ctx, ...);
4  vx_image gray  = vxCreateVirtualImage(...);
5  vx_image gauss = vxCreateVirtualImage(...);
6  vx_image gradX = vxCreateVirtualImage(...);
7  vx_image gradY = vxCreateVirtualImage(...);
8  vx_image mag   = vxCreateImage(ctx, ...);
9  vx_image phase = vxCreateImage(ctx, ...);
10
11 vx_graph graph = vxCreateGraph(context);
12
13 vxColorConvertNode(graph, rbg, gray);
14 vxGaussian3x3Node(graph, gray, gauss),
15 vxSobel3x3Node(graph, gauss, gradX, gradY);
16 vxMagnitudeNode(graph, gradX, gradY, mag);
17 vxPhaseNode(graph, gradX, gradY, phase);
18
19 status = vxVerifyGraph(graph);
20 if (status != VX_SUCCESS) abort();
21
22 while (/* input images? */) {
23   /* capture data into rgb */
24   vxProcessGraph(graph);
25   /* use data from out */
26 }
27
28 vxReleaseContext(&context);
```

Listing 1: OpenVX example

This introductory example follows these steps:

- A context is initially created (line 1) and then released at the end (line 28).
- Images are defined (lines 3-9), some of them as virtual (lines 4-7).
- A graph is created (line 11).
- A set of nodes is created and added to the graph as instances of vision kernels (lines 13-17).
- The `vxVerifyGraph` function (line 19) checks the graph consistency.
- The `vxProcessGraph` function (line 24) executes the graph in synchronous blocking mode inside a loop, that is a typical programming pattern to process an incoming stream of input images.

Overall, the OpenVX standard has been designed to support efficient execution on a wide range of architectures, while maintaining a consistent vision acceleration API for application portability. In this scenario, vendors should provide implementations for different hardware architectures, such as CPUs, GPUs. DSPs, FPGAs, and dedicated ASICs. The OpenVX high-level specification hides hardware details, but a vendor could implement a wide range of platform-specific optimization techniques.

III. Architectural Template

Figure 3 shows a block diagram of the architectural template targeted in this work. It consists of a general-purpose host processor coupled with a clustered many-core accelerator (CMA) inside an embedded system on chip (SoC) platform. The multi-cluster design is a common solution applied to overcome scalability limitations in modern many-core accelerators, such as STM STHORM [5], Plurality HAL [22], KALRAY MMPA [16], Adapteva Epiphany-IV [4] and PULP [10].

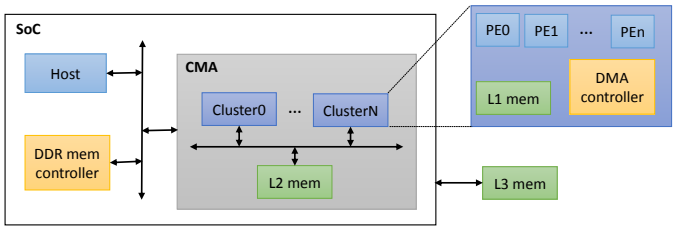

Fig. 3: Reference architectural template.

The processing elements (PEs) inside a cluster are fully independent RISC cores, supporting both SIMD and MIMD parallelism. Each PE is equipped with a private instruction cache. To avoid memory coherency overhead and increase energy efficiency, the PEs do not have private data caches. All PEs share a L1 multi-banked tightly coupled data memory (TCDM) acting as a scratchpad. The TCDM has a number of ports equal to the number of memory banks to provide concurrent access to different memory locations. Communication between the cores and the TCDM is based on a single-cycle logarithmic interconnect, implementing a word-level interleaving scheme to reduce the access contention to TCDM banks. The architectural template also includes a L2 scratchpad memory at SoC level and an external L3 DDR accessible by means of a memory controller. Both host cores and PEs can access the whole memory space, that is modeled as a partitioned global address space (PGAS). A DMA engine enables communication with other clusters, L2 memory and external peripherals.

The current version of ADRENALINE models a SoC with a single accelerator cluster, while future releases will evolve into a full multi-cluster environment. Our OpenVX run-time already supports the execution of OpenVX programs on multi-cluster accelerators by applying two methodologies: (i) the use of different clusters to compute different input sets, and (ii) the partition of an input set into multiple parts that can be computed independently.

IV. ADRENALINE INTERNALS

ADRENALINE comes with a *virtual platform* modeling the target architecture template and an *OpenVX run-time* optimized for many-core accelerators. This tool has been developed following two main objectives, application benchmarking/profiling and architecture tuning.

A. Virtual platform

The virtual platform is written in Python and C++. Python is used for the architecture instantiation and configuration, and also for the high-level execution management. C++ is used for implementing the models in an efficient manner, so that only binary code is called during normal execution.

A key functionality is the possibility of describing a block as a combinatory network. For instance, it is used to describe the interconnection between the cores and the TCDM. A library of basic components is available, but custom blocks can also be implemented and assembled.

In the following paragraphs we report some additional details about the most relevant blocks.

OpenRISC core. It is modeled with an Instruction Set Simulator (ISS) for the OpenRISC ISA [1], extended with timing modeling to consider the various sources of pipeline stalls. Adopting the same interface, an ISS for a different architecture can be plugged.

Memories. The memory blocks use a simple timing model, with a fixed latency for each reported access.

L1 interconnect. This block has an important impact on data accesses. The timings of an application can really differ depending on how the data buffers are accessed from memories. In our model, each target memory bank can accept one request per cycle, as provided by the architectural template.

Other interconnects. In the interconnect model a single request is never split, it traverses all the interfaces to the final target not allowing fine-grained arbitration. However there is a bandwidth model which is applied on each request, with the aim to report realistic timings.

DMA. The DMA sends a single synchronous request to the interconnect for each line to be transferred. The interconnect reports a latency for the whole transfer. As the DMA is able to fully stream several input requests while writing output requests, another input request can be scheduled after the latency of the first input request.

Shared instruction cache. The shared instruction cache model is made of an interconnect model and a set of cache banks. Each cache bank is a classic cache model that is able to report the latency to a request, depending on the fact it is a hit or a miss. The interconnect model is quite similar to the memory interconnect model with a support for multicast and a model of the L0 prefetch buffer. The L0 prefetch buffer is modeled as a classic cache with a single entry, so that only the misses are propagated to the L1 interconnect.

The current version supports the simulation of a single cluster and a single-core host. The host used for experiments is an OpenRISC core, with the aim to have comparable metrics for results. To instantiate a fully heterogeneous system, the host could be configured as an ARM or a x86 core. ADRENALINE provides the following tunable parameters:

- number of cores;
- hardware FPU enabled/disabled;
- available memory at L1, L2 and L3 levels;
- DMA bandwidth/latency.

ADRENALINE can be extended by defining new modules. A module is a Python class which declares the input and output ports that can be connected to other modules to specify the connections between the architectural blocks (e.g., a router to a memory). Then each Python class has a corresponding C++ class that implements the block model. When the platform is started, each C++ class receives all the configuration from the Python class (how ports are connected, property values), so that only C++ code is running during simulation.

Overall, writing a new module requires a limited effort to write the Python class, as it mainly contains declarations. Then the difficulty of writing the C++ model usually depends on the timing behaviour complexity of the block.

B. OpenVX run-time

The low-level programming environment in ADRENALINE is based on OpenCL 1.1 [17]. A common issue of using OpenCL on embedded systems is related to the mandatory use of global memory space to share intermediate data between kernels. When increasing the number of interacting kernels, the main memory bandwidth required to fulfill data requests originated by PEs is much higher than the available one, causing a bottleneck. To overcome this limitation, we extended OpenCL semantics to support explicit memory management. This feature enables more control and efficient reuse of on-chip memory, and greatly reduces the recourse to off-chip memory for storing intermediate results. Our OpenVX framework for CMA architectures is based on this extended OpenCL run-time, but it hides all the low-level

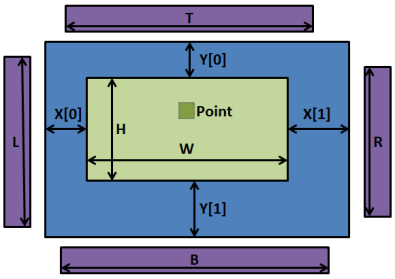

Fig. 4: Internal tiling descriptor

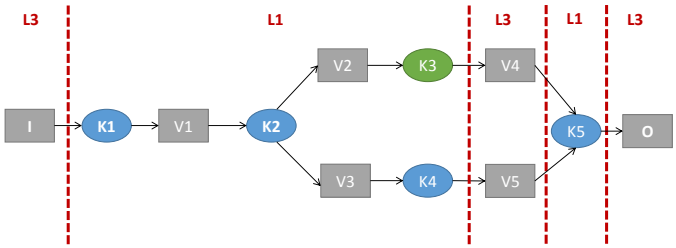

Fig. 5: Application graph partitioning

details behind a standard kernel interface, and even more important, it automatically handles partition and scheduling of data and kernels.

In our implementation we enforce *localized execution*, which implies that when a kernel is executed on a cluster, read/write operations are always performed on local buffers in the L1 scratchpad memory. The major advantage of this approach is that processor stalls due to reads/writes on L1 are very limited, as the access latency is low. Unfortunately, in real platforms the L1 is typically too small to contain a full image, moreover multiple kernels requires more L1 buffers to store intermediate results. As a natural solution to this problem, images are partitioned into smaller blocks, called *tiles*. With a proper sizing, multiple tiles can use different L1 buffers. In addition, the use of *double buffering* on input/output images enables to overlap between data transfers and computation, avoiding that cores are waiting during DMA transfers. Overall, the tile size strictly depends on the data access patterns used by kernels.

OpenVX standard does not specify which patterns should be supported and how they drive the run-time optimizations. Based on literature [25], we consider five remarkable classes of vision operators:

1) *Point operators* (e.g. color conversion, threshold) compute the value of each output point from the corresponding input point.
2) *Local neighbor operators* (e.g. linear operators, morphological operators) compute the value of a point in the output image that corresponds to the input tile.
3) *Recursive neighbor operators* (e.g. integral image) are similar to the previous ones, but in addition they also consider the previously computed values in the output tile.
4) *Global operators* (e.g. DFT) compute the value of a point in the output image using the whole input image.
5) *Geometric operators* (e.g. affine transforms) compute the value of a point in the output image using a non-rectangular input area.
6) *Statistical operators* (e.g. mean, histogram) compute statistical functions of image points.

To describe tiling patterns inside the run-time, we associate a *tiling descriptor* to each node parameter. Figure 4 describes a tiling descriptor. W and H are the dimensions of the *computing area*, that is the set of points used to compute a single output value; for output tiles, these values represent the minimum number of output points generated by a single computation. x and y values describe the neighboring area, that is the set of additional points contributing to the computation of a single output value, but they may belong to other computing or neighboring areas. In the framework we support three kernel classes:

- `CLE_KERNEL_NORMAL` is used for point and local neighbor operators. Tiling is used on both input and output images.
- `CLE_KERNEL_STAT` is used for statistical operators. Tiling can be activated just on input images, and we can use a persistent buffer to implement a reduction pattern "walking" through the tiles.
- `CLE_KERNEL_HOST` is used when it is impossible to apply tiling to input data, and this implies the kernel is executed by the host processor. This is true for global operators, and also in case of many geometrical operators we cannot apply a classical input tiling due to the irregular shape of the neighboring area.

The managing of tiling in recursive neighbor operators is equivalent to local neighbor operators, but we also need to save state data between tiles. For each kernel we provide a state buffer, using one of these policies:

- `CLE_STATE_NONE`: no state is required;
- `CLE_STATE_SCALAR`: an amount of state proportional to a scalar.
- `CLE_STATE_BORDER`: an amount of state proportional to the border size.
- `CLE_STATE_TILE`: an amount of state proportional to the whole tile size.

To support our run-time optimizations, we append additional steps to graph verification. These are the required steps, with a short description of the implementation that is currently provided:

- *Node scheduling* – a schedule is determined through a breadth-first visit, starting from the kernels connected to graph input data (head nodes). At each step a single kernel is selected for execution. For kernels which are executed on the accelerator, the code parallelism is exploited at node level.
- *Buffer allocation* – The initial number of L1 buffers is equals to the number of input images to the graph. For each kernel in the schedule list, the corresponding output buffers are allocated. A reference counting mechanisms is enabled, so a buffer can be reused if there is no further references to it.
- *Tile size propagation* – Overlap between adjacent tiles enforces data locality on buffers at the cost of transferring and/or computing the tile borders more than once. The minimum tile size is computed into two passes: a (i) forward pass, simulating a graph execution and simultaneously collecting the tiling constraints for each kernel, and a (ii) backward pass, starting from the last simulated node and setting the buffer final overlap according to all collected constraints
- *Buffer sizing* – Heuristic approach: Set the size of each buffer equal to its upper bound (full image size),

alternatively halve width and height for all buffers until the total memory footprint (i) fits the L1 available memory and (ii) is greater than the related lower bound (maximum tile size, including neighborhood).

- *Graph partitioning* – When the graph cannot be executed allocating all the intermediate tiles in L1 buffer, the graph is automatically partitioned into multiple sub-graphs.

At the end of graph verification stage, the framework derives an ordered partition set of the original OpenVX graph, each element consisting of a single host kernel or an OpenCL graph. At the execution stage single kernels are executed on the host, while OpenCL graphs are executed on the accelerator.

On the host side, we provide a set of kernels implemented on C using the OpenVX API to access data objects that has been introduced in Section II, based on the reference implementation provided by Khronos. On the accelerator side, we provide a set of OpenCL kernels which access input/output parameters directly addressing the global memory space, without explicit use of DMA primitives and intermediate local buffers. All the boilerplate code related to tiling and localized execution is managed by the run-time on the basis of kernel descriptors and graph verification steps.

Using our OpenVX run-time, the orchestration of multiple kernel nodes and accelerator sub-graphs is totally transparent to the programmer. For instance, in the application graph depicted in Figure 5 we suppose that K3 is a statistical kernel (e.g., a histogram). Tiling cannot be used on its output image V4, because each input tile contributes to sparse data in the result set (the histogram bins). Consequently, a memory boundary is added to its output, and this includes all the images read by K5. In this example K5 is a kernel of classes 1, 2 or 3, so a memory boundary is inserted to switch back its input images to L1 domain. To guarantee consistency in the host environment, I and O must reside on L3 domain.

In the most general case, the same result cannot be obtained by fusing kernels and optimally tiling the generated loop, as tiling requirements of different kernels are not homogeneous. Overall, the use of OpenVX graphs enables a global level of optimization which is not possible under a single-function paradigm

V. Experimental results

In this section we illustrate some use cases for ADRENALINE, and we show a comparison with a real platform.

A. Run-time optimizations

To assess the benefits of our run-time optimizations on common vision algorithms, we have taken into account the following benchmarks (see Table I):

- *Sobel* is a an edge detector based on Sobel filter;
- *FAST9* implements the FAST9 corner detection algorithm;
- *Disparity* computes the stereo-matching disparity between two images;
- *Pyramid* creates a set of scaled and blurred images (a pyramid);
- *Optical* implements the Lucas-Kanade algorithm to measure optical flow field;
- *Canny* implements the Canny edge detector algorithm.

Figure 6 shows the speed-up of the OpenVX accelerated versions compared to their OpenCL implementation. OpenCL applications are built using a library of image processing kernels, and kernel sources are manually optimized to support asynchronous data transfers and double buffering. In these tests

Benchmark	Nodes	Images (in/out/virtual)
Sobel	4	1 / 1 / 4
FAST9	3	1 / 1 / 3
Disparity	5	2 / 1 / 6
Pyramid	6	1 / 1 / 4
Optical	4	1 / 1 / 2
Canny	4	1 / 1 / 5

TABLE I: OpenVX benchmarks

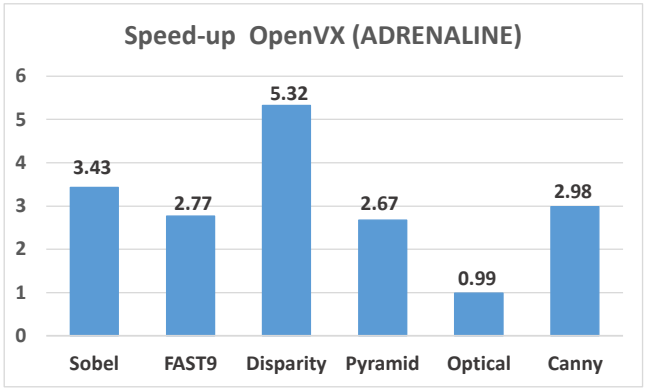

Fig. 6: Speed-up of OpenVX compared to OpenCL (ADRENALINE)

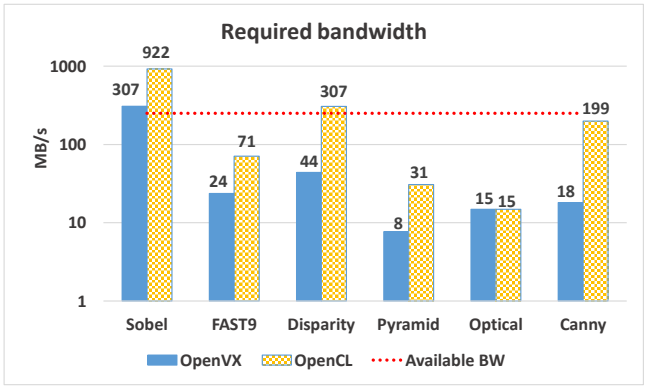

Fig. 7: Required bandwidth (OpenVX VS OpenCL)

we have considered this configuration for the virtual platform: 16 cores with hardware floating point support, 256KB L1 memory, 2MB L2 memory, 500MB L3 memory, 250MB/s for DMA bandwidth, 450 cycles for DMA latency. This configuration corresponds to an embedded system with severe bandwidth constraints. Moreover, all tests are performed with an input image size of 640×480 pixels. This size is sufficient to already see the effects of memory bandwidth, even larger benefits can be achieved for larger images.

Figure 7 reports the bandwidth requirements of the benchmarks for both OpenVX and OpenCL. These values are computed as the ratio between the total transferred bytes and the required computation time. The OpenVX version requires a lower bandwidth, with the exception of *Optical* that is implemented using a single kernel invoked multiple times. The bandwidth required by an the OpenCL application exceeds the available one (represented by the dashed line) in *Sobel* and *Disparity*, further limiting the speed-up of the OpenCL solution. Since each OpenCL kernel copies its outputs to L3 memory to pass data to the next one, the speed-up is closely

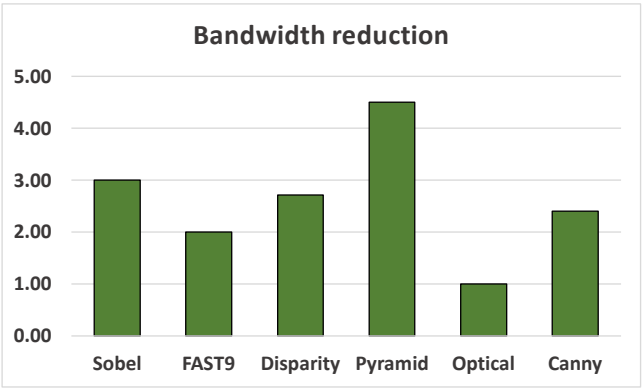

Fig. 8: Bandwidth ratio (OpenCL/OpenVX)

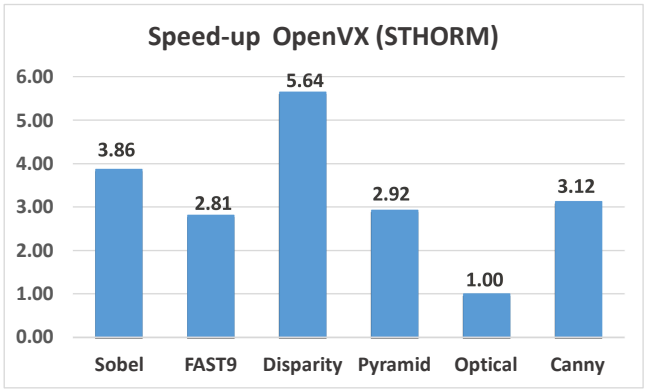

Fig. 9: Speed-up of OpenVX compared to OpenCL (STHORM board)

Benchmark	OpenCL	OpenVX
Sobel	703	423
Canny	475	335
Disparity	633	402
FAST9	428	348
Optical	123	83
Pyramid	441	321

TABLE II: Lines of code (OpenCL VS OpenVX)

related to the L3 bandwidth reduction. Figure 8 reports the ratio between the data transferred by the OpenVX version and the data transferred by the OpenCL version. It is evident that this ratio is strictly correlated to the speed-up. Table II reports the lines of code that a programmer must write to encode OpenVX or OpenCL kernels. In OpenCL kernels, programmers need to interleave DMA and computation explicitly, and the required lines of code are about 30% more than OpenVX ones. Moreover, these statistics do not take into account the boilerplate code that is required on the host side to setup and use OpenCL API.

B. Comparison with a real platform

To validate the virtual platform, we have compared ADRENALINE with a STHORM-based board. This board includes a Xilinx Zynq 7020 chip, featuring an ARM Cortex A9 dual core host processor operating at 667MHz plus FPGA programmable logic, and a STHORM many-core accelerator clocked at 430MHz. The ARM subsystem on the Zynq is connected to an AMBA AXI interconnection matrix, through which it accesses the DRAM controller. The latter is connected to the on-board DDR3 (500MB), which is the third memory level in the system (L3) for both ARM and STHORM cores. To allow transactions generated inside the STHORM chip to reach the L3 memory, and transactions generated inside the ARM system to reach internal STHORM L1 and L2 memories, part of the FPGA area is used to implement an *access bridge*.

The access bridge is clocked very conservatively at 40MHz; consequently, the main memory bandwidth available to the STHORM chip is limited to 250MB/s for the read channel and 125MB/s for the write channel, with an access latency of about 450 cycles. This configuration corresponds to the one used for experiments. Since the ISAs are different, we use a relative metric to compare the two solutions, that is the speed-up of OpenVX compared to OpenCL. Figure 9 reports the speed-up values measured on the STHORM platform. The comparison with Figure 6 shows that the speed-up trend is coherent, with an average margin of 6%. This implies that ADRENALINE models the bandwidth effects on a real embedded system with very good accuracy, even when the bandwidth is a severely constrained resource.

C. Application partitioning

As an example of partitioning optimization, we have considered a single application graph containing both a Canny edge detector and a FAST9 corner detector.

Figure 11 shows the graph representation of a Canny edge detector. This algorithm applies a Gaussian filter to de-noise the input image, and then apply a Sobel operator to find horizontal and vertical gradients. Starting from gradients, it computes the edges in terms of magnitude and phase. Finally, it performs *non-maxima suppression* to thin the edges and *hysteresis thresholding* to remove loosely connected points.

FAST9 corner detector [24] extracts corners by evaluating pixels on the Bresenham circle around a candidate point. If N contiguous pixels are brighter or darker than the central point by at least a threshold value then this point is considered to be a corner candidate. For each detection, its strength is computed and finally a non-maxima suppression step is applied to reduce the final points. The OpenVX standard library provides a specific node for FAST9 algorithm (`vxFastCornersNode`), which reads an input image and outputs an array of corner coordinates. This version differs from the one used in the previous sections, which is split into multiple kernels to run on the accelerator but provides as output a full image.

Figure 10 shows the timings resulting from the different mappings of FAST node w.r.t. Canny edge detector. For these specific implementations, the best solution is the one executing Canny on the accelerator and FAST9 on the host side. This is a non trivial result that fully highlights the benefits of this kind of analysis.

D. Architectural configuration

To evaluate different platform parameters, we have executed Canny edge detector measuring the execution time when the number of cores is set to increasing values. Figure 12 depicts the execution time in cycles. The bar on the left represents the time required to execute the whole application on the host using the reference implementation provided by Khronos. For this application, the lack of optimizations and the overhead due to OpenVX data access functions make more profitable the execution on the accelerator, even in case of a single core (1.58x).

Table III reports the speed-ups obtained using different configurations of the many-core accelerator. The efficiency decreases with the number of cores, and using this table a

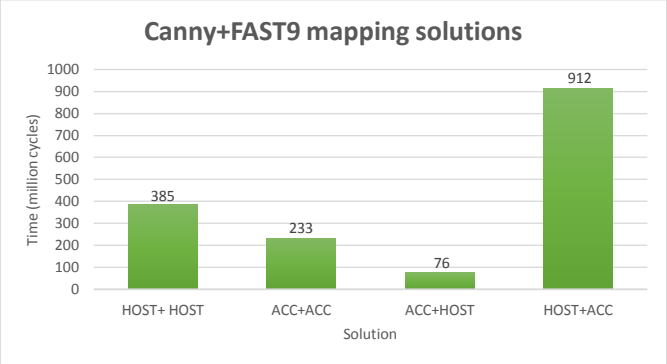

Fig. 10: Mapping of Canny and FAST9

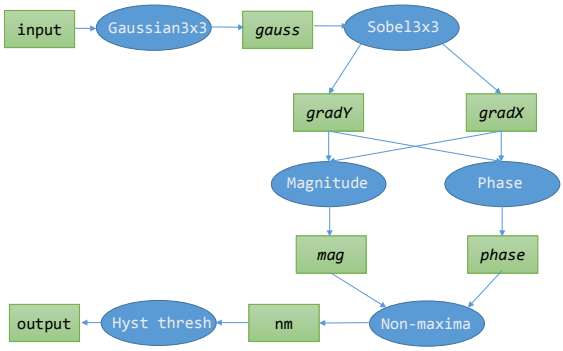

Fig. 11: Canny edge detector

designer can choose a valid trade-off. Table IV reports the effect of DMA latency on execution time, with a bandwidth set at 8 bytes/cycle. In this case our optimization approach based on localized execution and double buffering hides the variations of platform parameters related to DDR connection. In the most general case, this conditions can be verified for any application using our tool.

VI. Related Works

In the context of embedded vision systems, in the last years many FPGA-based solutions has been presented (e.g., [7] and [19], and [15]). We can also find several examples of domain-specific architectures. CHARM [8] and AXR-CMP [9] propose a framework for composable accelerators assembled

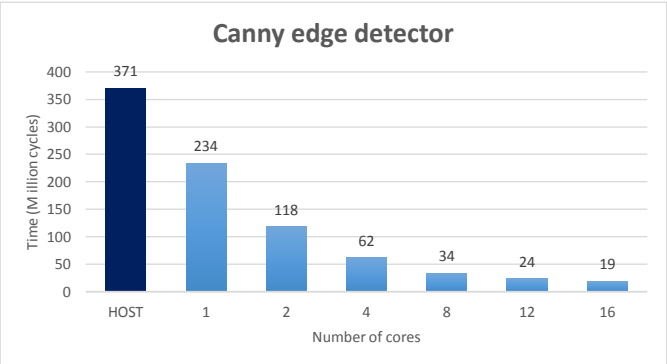

Fig. 12: Evaluation of Canny edge detector

Cores	Speed-up
1	1.00
2	1.98
4	3.79
8	6.97
12	9.70
16	12.00

TABLE III: Speed-up

Latency (cycles)	Execution time (Mcycles)
100	61.75
200	61.98
400	62.70

TABLE IV: Effect of DMA latency on execution time

from accelerator building blocks with dedicated DMA engines, while NeuFlow [14] is a special-purpose data-flow processors targeted for vision algorithms. Both FPGA and domain-specific architectures have emerged to satisfy demands for power-efficient and high-performance multiprocessing. Our solution is based on a general-purpose accelerator, using a programming model which takes into account the specific needs of CV domain but still supporting general-purpose programming (via standard or extended OpenCL).

Darkroom is a tool that synthesizes hardware descriptions for ASIC or FPGA, or optimized CPU code using an optimally scheduled pipeline. The tiling approach is similar to our solution, but there are some limitations. They consider just two access patterns (point-wise and stencil), that are the only to be compatible with the pipeline execution.

OpenCV [21] is an open-source and cross-platform library featuring high-level APIs for vision domain. OpenCV is the de-facto standard in desktop computing environment, its mainstream version is optimized for multi-core processors but it is not suitable for acceleration on embedded many-core systems. Some vendors provide accelerated versions of OpenCV which have been optimized for their hardware (e.g. OpenCV for Texas Instruments embedded platforms [11] or OpenCV for Tegra [27]). As a matter of fact, OpenCV needs a lower-level middleware for accelerating image processing primitives. This is precisely the goal of OpenVX, which aims at providing a standardized set of accelerated primitives, thereby enabling platform agnostic acceleration.

OpenCL [26] is a very widespread programming environment for both many-core accelerators and GPUs, and it is supported for an increasing number of heterogeneous architectures; for instance, Altera supports OpenCL on its FPGA architecture [12]. The OpenCL memory model is too constrained for SoC solutions, hence many extensions have been proposed by vendors. For instance, AMD provides a zero-copy mechanism to share data between host and GPU in Fusion APU products, also enabling the access to GPU local memory by host side through a unified north-bridge with full cache coherence [6]. In a many-core accelerator we need even more control on data allocation, because cores are not working in lock-step. In addition, we need the possibility to map the logical global space at different levels of the memory hierarchy, to efficiently maintain state between kernels.

Focusing on domain-specific approaches, Halide [23] is a programming language specifically designed to describe image processing pipelines with different architectural targets, including an OpenCL code generator. To implement an algorithm with Halide, the user must specify a functional description using a specific language. Halide defines a model based on

stencil pipelines, with the aim to find a trade-off between locality, exploitation of parallelism, and redundant re-computation. Even if the basic principles are the same, our framework is based on specific architectural features of a class of many-core accelerators, so we can simplify some allocation problem constraints with minimum loss of generality. Moreover, the functional model of Halide is not Turing-complete.

Graph-structured program abstractions have been studied for years in the context of streaming languages (e.g. StreamIt [28]). In these approaches, static graph analysis enables stream compilers to simultaneously optimize data locality by interleaving computation and communication between nodes. However, most research has focused on 1D streams, while image processing kernels can be modeled as programs on 2D and 3D streams. The model of computation required by image processing is also more constrained than general streams, because it is characterized by specific data access patterns.

Stencil kernels are a class of algorithms applied to multi-dimensional arrays, in which an output point is updated with weighted contributions from a subset of neighbor input points (called window or stencil). Our definition of tiles is equivalent to a 2D stencil. Many optimization techniques have been proposed to execute stencil kernels on multi-core platforms [13], but an effective solution for many-core accelerators executing heterogeneous vision kernels has not been proposed yet. Such a solution has to consider all the data access pattern specific of this domain, handling the possible overlapping of input windows and providing a solution for the access patterns that are not properly describable in terms of stencil computation.

KernelGenius [20] is a tool that enables the high-level description of vision kernels using a custom programming language. It aims at generating an optimized OpenCL kernel targeting the STHORM platform, with a totally transparent management of the DMA data transfers. The structure of the tiling problem for a single kernel is analogous to the formulation we are using in this work. The final performance are equivalent to an optimized OpenCL solution, but there are also the same limitations that we have reported for OpenCL.

VII. Conclusions

In this paper we have introduced ADRENALINE, a framework for fast prototyping and optimization of OpenVX applications that includes an OpenVX run-time and a virtual platform. The use cases proposed in Section V show how ADRENALINE can be used to highlight optimization opportunities for a wide range of end users, from hardware designers to CV researchers. In the near future we will also provide a new release of the framework supporting a multi-cluster platform. Moreover, next versions of the tool will support the execution on Xilinx FPGA boards, mapping the host on the ARM cores and the clusters on programmable logic, with the aim to further enhance simulation speed and accuracy.

VIII. Acknowledgements

This work has been supported by the EU-funded research projects MULTITHERMAN (g.a. 291125), PHIDIAS (g.a. 318013) and P-SOCRATES (g.a. 611016).

References

[1] OPENCORES Project: OpenRISC 1000 processor. http://www.opencores.org/projects/or1k/. Accessed: 2014-08-31.

[2] The Khronos Group Inc. http://www.khronos.org.

[3] The PULP Project. http://www-micrel.deis.unibo.it/pulp-project.

[4] Adapteva, Inc. Epiphany-IV 64-core 28nm Microprocessor. http://www.adapteva.com/products/silicon-devices/e64g401/.

[5] L. Benini, E. Flamand, D. Fuin, and D. Melpignano. P2012: Building an ecosystem for a scalable, modular and high-efficiency embedded computing accelerator. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2012*. IEEE, 2012.

[6] P. Boudier and G. Sellers. Memory System on Fusion APUs. *AMD fusion developer summit*, 2011.

[7] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown, and T. Czajkowski. LegUp: high-level synthesis for FPGA-based processor/accelerator systems. In *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*. ACM, 2011.

[8] J. Cong, M. A. Ghodrat, M. Gill, B. Grigorian, and G. Reinman. CHARM: A Composable Heterogeneous Accelerator-rich Microprocessor. In *Proceedings of the 2012 ACM/IEEE International Symposium on Low Power Electronics and Design*, ISLPED '12. ACM, 2012.

[9] J. Cong, C. Liu, M. A. Ghodrat, G. Reinman, M. Gill, and Y. Zou. AXR-CMP: Architecture Support in Accelerator-Rich CMPs.

[10] F. Conti, D. Rossi, A. Pullini, I. Loi, and L. Benini. Energy-efficient vision on the PULP platform for ultra-low power parallel computing. In *2014 IEEE Workshop on Signal Processing Systems (SiPS)*, 2014.

[11] J. Coombs and R. Prabhu. OpenCV on TIs DSP+ ARM® platforms: Mitigating the challenges of porting OpenCV to embedded platforms. *Texas Instruments*, 2011.

[12] T. S. Czajkowski, U. Aydonat, D. Denisenko, J. Freeman, M. Kinsner, D. Neto, J. Wong, P. Yiannacouras, and D. P. Singh. From OpenCL to high-performance hardware on FPGAs. In *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on*. IEEE, 2012.

[13] K. Datta, S. Kamil, S. Williams, L. Oliker, J. Shalf, and K. Yelick. Optimization and performance modeling of stencil computations on modern microprocessors. *SIAM review*, 2009.

[14] C. Farabet, B. Martini, B. Corda, P. Akselrod, E. Culurciello, and Y. LeCun. Neuflow: A runtime reconfigurable dataflow processor for vision. In *Computer Vision and Pattern Recognition Workshops (CVPRW), 2011 IEEE Computer Society Conference on*. IEEE, 2011.

[15] S. K. Gehrig, F. Eberli, and T. Meyer. A real-time low-power stereo vision engine using semi-global matching. In *Computer Vision Systems*. Springer, 2009.

[16] KALRAY Corporation. http://www.kalray.eu/.

[17] Kronos Group. The OpenCL 1.1 Specifications. http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf, 2010.

[18] Kronos Group. The OpenVX API for hardware acceleration. http://www.khronos.org/openvx, 2014.

[19] Y. Lei, Z. Gang, R. Si-Heon, L. Choon-Young, L. Sang-Ryong, and K.-M. Bae. The platform of image acquisition and processing system based on DSP and FPGA. In *Smart Manufacturing Application, 2008. ICSMA 2008. International Conference on*. IEEE, 2008.

[20] T. Lepley, P. Paulin, and E. Flamand. A Novel Compilation Approach for Image Processing Graphs on a Many-core Platform with Explicitly Managed Memory. In *Proceedings of the 2013 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*. IEEE, 2013.

[21] OpenCV Library Homepage. http://www.opencv.com/.

[22] Plurality Ltd. The HyperCore Processor. http://www.plurality.com/hypercore.html.

[23] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*. ACM, 2013.

[24] E. Rosten and T. Drummond. Machine learning for high-speed corner detection. In *Computer Vision–ECCV 2006*. Springer, 2006.

[25] M. Sonka, V. Hlavac, R. Boyle, et al. *Image processing, analysis, and machine vision*. Thomson Toronto, 2008.

[26] J. E. Stone, D. Gohara, and G. Shi. OpenCL: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 2010.

[27] Tegra Android Development Documentation Website. http://docs.nvidia.com/tegra/index.html.

[28] W. Thies, M. Karczmarek, and S. Amarasinghe. StreamIt: A language for streaming applications. In *Compiler Construction*. Springer, 2002.